Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ

по лабораторной работе №3

на тему

ВВЕДЕНИЕ В ТЕХНОЛОГИЮ CUDA

Вариант 2.1

Студенты: Е.А. Петрович

М.А. Ходосевич

Преподаватель: Т.С. Жук

МИНСК 2024

**1 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

**1.1 Цель работы**

Изучить особенности написания программ на с использованием CUDA. Изучить и сравнить производительность алгоритмов перестановки элементов матрицы на центральном и графическом процессоре с использованием технологии CUDA.

**1.2 Исходные данные к работе**

В исходной матрице NxM, используя окно, переставить элементы с шагом N/2 и M/2. Размер выходной матрицы меняется пропорционально в зависимости от заданного окна. Входное окно согласно варианту: 2х2. Тип данных – int.

Требования:

1. Реализация CPU и GPU.

2. Сравнение времени работы в едином формате (микро-, миллисекунды и т.п.). Измерение времени на GPU через события в CUDA.

3. Сравнение результатов работы: полное поэлементное сравнения массива и вывод фрагмента на экран.

4. Допускается использование двух массивов: с входными и выходными данными. Дополнительные массивы запрещены.

5. Инициализация массива случайными числами.

6. Проверка ошибок выполнения cudaError\_t.

Для сдачи лабораторной работы достаточно просто переместить элементы, но можно сделать дополнительную оптимизацию.

Каждая оптимизация – отдельное ядро, основанное на предыдущей реализации. Исключение – разный размер матриц. Без транзакций в глобальную память – самый простой вариант. Можно добавить: работа с матрицами, не кратными размеру блока и размерам обрабатываемого фрагмента, корректный доступ в глобальную память, разделяемая память и корректное обращение по банкам памяти, применение технологии CUDA Stream.

**2 ВЫПОЛНЕНИЕ РАБОТЫ**

В ходе экспериментов изучалась производительность графического и центрального процессора на разных размерах матриц. Основной задачей лабораторной является перестановка элементов матрицы и замер времени выполнения.

Предполагалось, что при небольших размерах матриц работа центрального процессора будет занимать меньше времени. Также, при повторном запуске с теми же матрицами центральный процессор должен отрабатывать быстрее предыдущего из-за занесения данных в кеш. При увеличении размера матриц время работы центрального процессора должно увеличиваться и в какой-то момент стать больше времени работы графического процессора. То есть, предполагается, что на небольших размерах матрицы центральный процессор будет производительнее, на больших – производительнее графический процессор.

В таблицу 2.1 сведены время перестановки элементов матрицы для центрального (CPU) и графического (GPU) процессоров на различных матрицах.

Таблица 2.1 – Результаты работы

|  |  |  |  |
| --- | --- | --- | --- |
| Матрица | CPU, мс | GPU, мс | CPU/GPU |
| 10х10 | 0.003 | 0.6707 | 0.0045 |
| 100х100 | 0.1928 | 0.7014 | 0.1352 |
| 250х250 | 1.2528 | 0.9851 | 1.2341 |
| 500х500 | 4.093 | 1.3655 | 4.2316 |
| 1000х1000 | 11.7319 | 3.6291 | 3.2328 |
| 5000х5000 | 303.084 | 74.2973 | 4.0793 |
| 10000х10000 | 1222.67 | 318.426 | 3.8397 |
| 20000х20000 | 19860.3 | 1810.7 | 10.9683 |

Теория о том, что на больших матрицах время работы графического процессора будет меньше, чем центрального, и наоборот, подтвердилась. Это связано с хороших уровнем параллелизма и эффективным распределением вычислений по ядрам графического процессора

**3 ВЫВОДЫ**

В ходе лабораторной работы были изучены особенности написание программы с использованием CUDA. Были написаны реализации как на CPU, так и на GPU. Были подтверждены предполагаемые исходы работы программы. Также были получены навыки по установке требуемого ПО и работе на CUDA. Удалось написать работоспособную программу, которая удовлетворяет всем поставленным требованиям. По окончании выполнения лабораторной работы было доказано, что на больших размерах матриц намного эффективнее и производительнее будет использовать графический процессов.

**ПРИЛОЖЕНИЕ А**

(обязательное)

Исходный текст программы

Содержимое файла kernel.cu:

#include <iostream>

#include <cuda\_runtime.h>

#include <curand.h>

#include <curand\_kernel.h>

#include <vector>

#include <random>

#include <chrono>

// Вспомогательная функция для проверки ошибок CUDA

void checkCudaError(cudaError\_t err, const char\* msg) {

if (err != cudaSuccess) {

std::cerr << "CUDA Error (" << msg << "): " << cudaGetErrorString(err) << std::endl;

exit(EXIT\_FAILURE);

}

}

// Функция для создания матрицы с случайными значениями

std::vector<std::vector<int>> create\_random\_matrix(int rows, int cols) {

std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols));

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, 100); // Случайные значения от 0 до 100

for (int i = 0; i < rows; ++i)

for (int j = 0; j < cols; ++j)

matrix[i][j] = dis(gen);

return matrix;

}

// Ядро CUDA для перестановки элементов с шагом N/2 и M/2

\_\_global\_\_ void rearrange\_gpu(const int\* matrix, int\* result, int rows, int cols) {

int half\_rows = rows / 2;

int half\_cols = cols / 2;

int i = (blockIdx.y \* blockDim.y + threadIdx.y) \* 2;

int j = (blockIdx.x \* blockDim.x + threadIdx.x) \* 2;

if (i + 1 < rows && j + 1 < cols) {

// Перестановка элементов из блока 2x2

result[(i / 2) \* cols + (j / 2)] = matrix[i \* cols + j];

result[(i / 2) \* cols + (j / 2 + half\_cols)] = matrix[i \* cols + j + 1];

result[((i / 2) + half\_rows) \* cols + (j / 2)] = matrix[(i + 1) \* cols + j];

result[((i / 2) + half\_rows) \* cols + (j / 2 + half\_cols)] = matrix[(i + 1) \* cols + j + 1];

}

}

// Функция для вывода части матрицы

void print\_partial\_matrix(const std::vector<std::vector<int>>& matrix, int N) {

for (int i = 0; i < matrix.size() && i < N; ++i) {

for (int j = 0; j < matrix[0].size() && j < N; ++j) {

std::cout << matrix[i][j] << " ";

}

std::cout << std::endl;

}

}

// Функция для вывода части одномерного массива как двумерной матрицы

void print\_partial\_flat\_matrix(const std::vector<int>& matrix, int rows, int cols, int N) {

for (int i = 0; i < rows && i < N; ++i) {

for (int j = 0; j < cols && j < N; ++j) {

std::cout << matrix[i \* cols + j] << " ";

}

std::cout << std::endl;

}

}

// Функция для замера времени выполнения на CPU

void measure\_cpu(const std::vector<std::vector<int>>& matrix, std::vector<std::vector<int>>& result) {

int half\_rows = matrix.size() / 2;

int half\_cols = matrix[0].size() / 2;

auto start = std::chrono::high\_resolution\_clock::now();

// Перестановка элементов с шагом N/2 и M/2 на CPU

for (int i = 0; i < half\_rows \* 2; i += 2) {

for (int j = 0; j < half\_cols \* 2; j += 2) {

result[i / 2][j / 2] = matrix[i][j];

result[i / 2][j / 2 + half\_cols] = matrix[i][j + 1];

result[i / 2 + half\_rows][j / 2] = matrix[i + 1][j];

result[i / 2 + half\_rows][j / 2 + half\_cols] = matrix[i + 1][j + 1];

}

}

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<float> duration = end - start;

std::cout << "\nCPU Time: " << duration.count() \* 1000 << " milliseconds" << std::endl; // вывод времени в миллисекундах

}

// Функция для замера времени выполнения на GPU

void measure\_gpu(const int\* d\_matrix, int\* d\_result\_matrix, int rows, int cols) {

int threads\_per\_block = 32;

dim3 threads(threads\_per\_block, threads\_per\_block);

dim3 blocks((cols + threads\_per\_block \* 2 - 1) / (threads\_per\_block \* 2),

(rows + threads\_per\_block \* 2 - 1) / (threads\_per\_block \* 2));

cudaEvent\_t start, stop;

checkCudaError(cudaEventCreate(&start), "cudaEventCreate (start)");

checkCudaError(cudaEventCreate(&stop), "cudaEventCreate (stop)");

checkCudaError(cudaEventRecord(start), "cudaEventRecord (start)");

// Запуск CUDA ядра

rearrange\_gpu << <blocks, threads >> > (d\_matrix, d\_result\_matrix, rows, cols);

checkCudaError(cudaGetLastError(), "Kernel execution"); // Проверка на ошибку ядра

checkCudaError(cudaDeviceSynchronize(), "cudaDeviceSynchronize");

checkCudaError(cudaEventRecord(stop), "cudaEventRecord (stop)");

checkCudaError(cudaEventSynchronize(stop), "cudaEventSynchronize");

float milliseconds = 0;

checkCudaError(cudaEventElapsedTime(&milliseconds, start, stop), "cudaEventElapsedTime");

std::cout << "\nGPU Time: " << milliseconds << " milliseconds" << std::endl; // вывод времени в миллисекундах

checkCudaError(cudaEventDestroy(start), "cudaEventDestroy (start)");

checkCudaError(cudaEventDestroy(stop), "cudaEventDestroy (stop)");

}

// Полное поэлементное сравнение матриц

bool compare\_results(const std::vector<std::vector<int>>& cpu\_matrix, const std::vector<int>& gpu\_matrix, int rows, int cols) {

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

if (abs(cpu\_matrix[i][j] - gpu\_matrix[i \* cols + j]) > 1) { // Учитываем, что тип int

return false;

}

}

}

return true;

}

int main() {

int rows = 10000, cols = 10000;

size\_t matrix\_size = rows \* cols \* sizeof(int);

auto cpu\_matrix = create\_random\_matrix(rows, cols);

std::cout << "Original Matrix (first 10x10):" << std::endl;

print\_partial\_matrix(cpu\_matrix, 10);

// Копирование данных в линейный массив для GPU

std::vector<int> flat\_cpu\_matrix(rows \* cols);

for (int i = 0; i < rows; ++i)

std::copy(cpu\_matrix[i].begin(), cpu\_matrix[i].end(), flat\_cpu\_matrix.begin() + i \* cols);

int\* d\_matrix;

int\* d\_matrix\_res;

checkCudaError(cudaMalloc(&d\_matrix, matrix\_size), "cudaMalloc (d\_matrix)");

checkCudaError(cudaMalloc(&d\_matrix\_res, matrix\_size), "cudaMalloc (d\_matrix\_res)");

checkCudaError(cudaMemcpy(d\_matrix, flat\_cpu\_matrix.data(), matrix\_size, cudaMemcpyHostToDevice), "cudaMemcpy HostToDevice");

std::vector<std::vector<int>> cpu\_result(rows, std::vector<int>(cols));

measure\_cpu(cpu\_matrix, cpu\_result);

measure\_gpu(d\_matrix, d\_matrix\_res, rows, cols);

std::vector<int> gpu\_result(rows \* cols);

checkCudaError(cudaMemcpy(gpu\_result.data(), d\_matrix\_res, matrix\_size, cudaMemcpyDeviceToHost), "cudaMemcpy DeviceToHost");

if (compare\_results(cpu\_result, gpu\_result, rows, cols)) {

std::cout << "CPU and GPU results match!" << std::endl;

}

else {

std::cout << "Results differ!" << std::endl;

}

std::cout << "\nPartial Matrix CPU (first 10x10):" << std::endl;

print\_partial\_matrix(cpu\_result, 10);

std::cout << "\n\nPartial Matrix GPU (first 10x10):" << std::endl;

print\_partial\_flat\_matrix(gpu\_result, rows, cols, 10);

checkCudaError(cudaFree(d\_matrix), "cudaFree (d\_matrix)");

checkCudaError(cudaFree(d\_matrix\_res), "cudaFree (d\_matrix\_res)");

return 0;

}